An Incomplete Splitting-up Conjugate Gradient Method for Parallel Computing

Akiyoshi Wakatani
Faculty of Informatics and Intelligence, Konan University, Kobe, Japan

Abstract: This paper describes several variants of SPCG (splitting up conjugate gradient) method suitable for parallel computing and evaluates the performance and the speed of convergence on a distributed-memory multicomputer. SP (splitting-up) preconditioner can be easily parallelized because other dimensions except for one dimension are independent. Among the variants, one of incomplete SPCG method, which does not carry out one of three tridiagonal matrix solvers, achieves the best performance, and this method is about 20 times faster than one-process version of the SPCG method on 32 CPU cores of the multicomputer.

Key words: Iterative methods, tridiagonal matrix solver, preconditioning.

1. Introduction

Natural phenomena, such as weather prediction and fluid dynamics, can be formalized by appropriate partial differential equations. These equations are usually difficult to be solved analytically, but the numerical solutions for them can be found by a large scale numerical simulation with high-speed computers. Such a simulation can be usually solved by not direct methods, but iterative methods. Iterative methods include stationary iterative methods such as SOR (successive over-relaxation) method, and non-stationary iterative methods based on Krylov subspace method such as CG (conjugate gradient) method and GMRES method [1]. In recent years, the non-stationary iterative methods are frequently used because of their rapid convergence. However, since a high-speed computation using parallel processing is indispensable in order to achieve a large size simulation, we have to select an iterative method that is easy to be parallelized with keeping the convergence rate high. Usually, ICCG (incomplete Cholesky conjugate gradient) method, which is based on modified Cholesky decomposition, is used, but it is difficult to parallelize the method straightforwardly.

In this paper, we focus on a preconditioned CG method that is one of non-stationary iterative methods based Krylov subspace method. Especially, we deal with a CG method having SP (splitting-up) as a preconditioner, which is suitable for parallel computing. We also compare variants of the SPCG method that can be easily implemented on distributed-memory multicomputers.

The rest of this paper is organized as follows: Section II presents the SPCG method for solving Laplace equation and Section III proposes the implementation policy for distributed memory systems. Section IV presents the experimental method and discusses the results and Section V concludes this paper with a summary.

2. Splitting-up Conjugate Gradient Method

The speed of convergence of iterative methods depends on a condition number of a matrix that should be solved. In general, when the condition number is close to 1, the speed of convergence is fast. Therefore, in order to improve the speed of convergence of the CG method, the preconditioning is applied to a matrix that should be solved. Usually, the
ICCG method is used for solving a symmetric positive-definite matrix.

Although the speed of convergence of the ICCG method is fast, it is difficult to parallelize the method straightforwardly, and thus several variants have been studied for parallel processing. For example, by using a red-black ordering and a large-numbered multi-color ordering, the ICCG method can be parallelized, but the speed of convergence may be degraded [2]. Therefore, it is general that the parallelization of the ICCG method may result in the degradation of the convergence performance, although the computational performance can be improved.

On the other hand, SPCG (splitting-up conjugate gradient) method is a preconditioned CG method like the ICCG method, but the parallelization of the SPCG method is relatively easy and straightforward. According to the literature [3], it is known that the speed of convergence of the SPCG method is almost equal to that of the ICCG method.

Fig. 1 shows SPCG method that applies to the following three-dimensional Laplace equation: $ Au = f $ ( $ Au_{i,j,k} = 6u_{i,j,k} - u_{i-1,j,k} - u_{i,j,k-1} - u_{i+1,j,k} - u_{i,j,k+1} $)

where, $ u^n $ is an unknown array at the $ n $-th iteration, $ \tau $ and $ \beta $ are scalar parameters for the iterative method, $ h^n $ is a residual vector and $ d^n $ is a direction vector that is used for updating the unknown array.

\[
\begin{align*}
  h^n &= C^{-1}(Au^n - f) \\
  \tau &= \frac{(h^n, Ch^n)}{(d^n, Ad^n)} \\
  u^{n+1} &= u^n + \tau d^n \\
  h^n &= h^n + \tau C^{-1}Ad^n \\
  \beta &= \frac{(h^{n+1}, Ch^{n+1})}{(h^n, Ch^n)} \\
  d^{n+1} &= -h^{n+1} + \beta d^n
\end{align*}
\]

The preconditioner (splitting-up part) solves three tridiagonal matrices for three dimensions in the case of three dimensional Laplace equation. Namely, $(D + X)^{-1}, (D + Y)^{-1}$ and $(D + Z)^{-1}$ are tridiagonal matrix solvers for $x$-direction, $y$-direction and $z$-direction, respectively. Therefore, the solver for $(D + X)^{-1}$ can be parallelized in the $y$-direction and $z$-direction, the solver for $(D + Y)^{-1}$ can be parallelized in the $z$-direction and $x$-direction and the solver for $(D + Z)^{-1}$ can be parallelized in the $x$-direction and $y$-direction. In addition, a tridiagonal matrix is solved by Thomas method, which consists of a forward substitution and a backward substitution. A tridiagonal matrix equation is as follows:

\[
\begin{align*}
  b_0x_0 + c_0x_1 &= d_0 \\
  a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= d_i \quad (i = 1 \ldots N - 2) \\
  a_{N-1}x_{N-2} + b_{N-1}x_{N-1} &= d_{N-1}
\end{align*}
\]

Fig. 1  Splitting up CG method.
where \(a\), \(b\), \(c\) and \(d\) are given in advance and \(x\) is an unknown vector. By using auxiliary vectors \(l\) and \(m\), a forward substitution is as follows:

\[
l_0 = -\frac{c_0}{b_0}, m_0 = -\frac{d_0}{b_0}
\]

\[
l_i = -\frac{c_i}{a_i l_{i-1} + b_i} \quad (i = 1 \ldots N - 2)
\]

\[
m_i = \frac{d_i - a_i m_{i-1}}{a_i l_{i-1} + b_i} \quad (i = 1 \ldots N - 2)
\]

On the other hand, a backward substitution is as follows:

\[
x_{N-1} = \frac{d_{N-1} - a_{N-1} m_{N-2}}{a_{N-1} l_{N-2} + b_{N-1}}
\]

\[
x_i = l_i x_{i+1} + m_i \quad (i = N - 2 \ldots 0)
\]

It should be noted that these substitutions are carried out for adjacent array elements.

In order to distribute three-dimensional arrays on distributed-memory multicomputers, at least one dimension of three dimensions must be distributed. For example, when the distributed dimension is \(z\)-direction, tridiagonal matrix solvers in \(x\)-direction and in \(y\)-direction can be carried out without communications because all the data in \(x\)-direction and in \(y\)-direction are stored in a local memory, but a tridiagonal matrix solver in \(z\)-direction needs communications because the data in \(z\)-direction are distributed over processors.

### 3.2 Variants

1. **Array redistribution**: Array redistribution, which transforms the configuration of data distribution, is a simple method to implement communications in \(z\)-direction (SPCG (redist)). For example, when an array distributed in \(z\)-direction is redistributed in \(y\)-direction, Thomas method in \(z\)-direction can be done without communications because all the data in \(z\)-direction are stored in a local memory. However, since array redistributions must be carried out before and after the tridiagonal matrix solver, the overhead of the communication must be considered. It should be noted that the total computational complexity of Thomas method is \(O(N^3 / P)\), and the communication cost is \(O(N^3)\) where \(P\) is the number of processors.

2. **No array redistribution**: We have two methods without array redistributions to implement the SPCG method in parallel. First of all, a pipelining is utilized in Thomas method in \(z\)-direction (SPCG (pipe)). As mentioned, Thomas method consist of a forward substitution and a backward substitution. For example, when \(i = 1\) and \(j = 1\), a processor with rank \(k\) carries out the forward substitution using data that this processor has, and then this processor sends the last data to a processor with rank \(k+1\). After that, the processor with rank \(k+1\) carries out the forward substitution by using the received data, and the processor with rank \(k+1\) carries out the forward substitution when \(i = 1\) and \(j = 2\). Thus these processors run in concurrent, and other processors can run concurrently as well. The backward substitution is processed in the same way. However, in order to reduce the number of communications, several communications must be coalesced. So, in this paper, all the data in \(y\)-direction are communicated together, and the data in \(x\)-direction are processed in a pipeline manner.

Another method utilizes P-scheme [4] method that authors have proposed for solving a tridiagonal matrix without array redistributions (SPCG (p-scheme)). This method requires just three one-to-one communications between neighboring processors, but the computational complexity is a few times larger than the original SPCG method. However, since the communication time is usually dominant on multicomputers, this method is effective on distributed-memory systems. The tradeoff of computation and communication will be evaluated later. It should be noted that the total computational complexity is \(O(N^3 / P)\), and the communication cost is \(O(N^2 \times P)\). In general, \(O(N^2 \times P)\) is less
than $O(N^3)$. (3) Incomplete SPCG: The last variants do not utilize communications in order to improve a performance, although they deviate from the original SPCG method. Because SP preconditioner incompletely solves an original problem by nature, it is not so significant that the incompleteness of the preconditioner increases a little bit. So, the first method does not carry out the tridiagonal matrix solver in z-direction (NoZsolve). Since this does not solve in z-direction, the elapsed time can be reduced, and the communication time can be also alleviated, but a speed of convergence may be degraded. Therefore, the convergence performance should be evaluated by experiments, and thus a total performance, which is a combination of a speed of convergence and a computational speedup, should be also considered.

Next, the second method incompletely solves the tridiagonal matrix in z-direction (ISPCG (incomplete SPCG)). Namely, like SPCG (pipe), this method solves the matrix in z-direction, but this does not send the last value of the forward substitution to the neighboring processor. Since this method is closer to the original SPCG method than NoZSolve, a speed of convergence may be superior to NoZSolve, but the computational complexity increases a little bit. For this method, a total performance should be also considered by combining a speed of convergence with a computational speed.

### 4. Evaluation

We evaluate our approach on SX-ACE (NEC) super computers installed at Osaka University (Japan). SX-ACE consists of 512 nodes that contains 4 vector CPUs and 64 GB memory (256 GB/s), and the data are transferred between the nodes at the speed of 8 GB/s. The theoretical peak performance of the CPU is 64 GFLOPS, so the total performance reaches 132 TFLOPS [5].

#### 4.1 Speedups

Table 1 shows elapsed time of 100 iterations and speedups of several variants of the SPCG method in the case of the data size of 192$^3$ and 96$^3$.

<table>
<thead>
<tr>
<th></th>
<th>96x96×96 (sec)</th>
<th>192×192×192 (sec)</th>
<th>Speedup</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 proc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>Speedup</td>
<td>Time</td>
<td>Speedup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 proc</td>
<td>SPCG</td>
<td>0.63</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPCG (redist)</td>
<td>0.2388</td>
<td>2.64</td>
<td>1.94</td>
</tr>
<tr>
<td></td>
<td>NoZSolve</td>
<td>0.055</td>
<td>11.45</td>
<td>0.73</td>
</tr>
<tr>
<td>16 procs</td>
<td>SPCG (pipe)</td>
<td>0.32</td>
<td>1.97</td>
<td>2.85</td>
</tr>
<tr>
<td></td>
<td>ISPCG</td>
<td>0.067</td>
<td>9.40</td>
<td>0.852</td>
</tr>
<tr>
<td></td>
<td>SPCG (P-scheme)</td>
<td>0.13</td>
<td>4.85</td>
<td>2.32</td>
</tr>
<tr>
<td>32 procs</td>
<td>SPCG (redist)</td>
<td>0.27</td>
<td>2.33</td>
<td>1.437</td>
</tr>
<tr>
<td></td>
<td>NoZSolve</td>
<td>0.03659</td>
<td>17.22</td>
<td>0.377</td>
</tr>
<tr>
<td></td>
<td>SPCG (pipe)</td>
<td>0.353</td>
<td>1.78</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>ISPCG</td>
<td>0.04348</td>
<td>14.49</td>
<td>0.416</td>
</tr>
<tr>
<td></td>
<td>SPCG (P-scheme)</td>
<td>0.1795</td>
<td>3.51</td>
<td>0.964</td>
</tr>
</tbody>
</table>
An Incomplete Splitting-up Conjugate Gradient Method for Parallel Computing

data size, the number of communication of SPCG (p-scheme) is constant, so the speedup of SPCG (p-scheme) overcomes other SPCG methods because the communication cost of SPCG (p-scheme) is relatively small.

On the other hand, NoZSolve is faster than ISPCG, and much faster than three SPCG variants. For example, when the data size is $96^3$ and the number of processes is 32, the speedups of NoZSolv and ISPCG are 17.16 and 18.94, but the speedups of SPCG (redist), SPCG (pipe) and SPCG (p-scheme) are 4.97, 5.95 and 7.41, so the difference is about twofold. The reason is that both NoZSolv and ISPCG have no communications and NoZSolve does not include a solver in z-direction. However, a speed of convergence may be degraded although the computational performance is excellent. We will discuss this issue later. NoZSolve achieves a speedup of about 19 when the number of processes is 32. The reason why the best speedup is not close to 32 is that the CG method includes a matrix-vector product calculation that contains communications between processors.

4.2 Speed of Convergence

Fig. 2 shows speeds of convergence of the variants of the SPCG method. It should be noted that the speeds of convergence of SPCG (redist), SPCG (pipe) and SPCG (p-scheme) are same. Note that in the case of ISPCG, a speed of convergence depends on the number of processes.

![Speed of convergence](image-url)
As mentioned before, both NoZSolve and ISPCG generate different residuals from the original SPCG method. In the case of the data size of 96³, the trend of convergence of NoZSolve is almost same as the SPCG method until 70 iterations, but after 70 iterations, NoZSolve becomes worse than the SPCG method. The trend of convergence of ISPCG is worse than NoZSolve until 50 iterations, but after 80 iterations, ISPCG is superior to NoZSolve.

When the data size is 192³, the trend of convergence is different. The trend of NoZSolve is almost same as the SPCG method. The trend of convergence of ISPCG is worse than NoZSolve until 80 iterations, but after 80 iterations, ISPCG is almost same as NoZSolve.

In both cases, ISPCG does not overcome NoZSolve from the point of view of convergence. Since ISPCG is closer to the original SPCG method as a preconditioner, the convergence performance of ISPCG is expected to be superior to NoZSolve, but the result is opposite. This result may depend on the property of an object problem, but NoZSolve is comparable to the SPCG method.

4.3 Total Performance

By combining of the speed of convergence and the elapsed time of one iteration, a total performance can be calculated. Namely, a total performance is an elapsed time that is required for achieving a given value of residual for the variants of the SPCG method. Fig. 3 shows the total performance when the data size is 96³. Note that the number of processes is 32 (8 nodes), and SPCG (1 proc) shows the result of 1 process (1 CPU core).

As a whole, NoZSolve exhibits the best performance. For example, NoZSolve requires about 2 msec to reduce the residual to $1.0 \times 10^{-10}$, but SPCG (1 proc) needs about 38.5 msec, so the speedup is about 20. When the residual is $1.0 \times 10^{-10}$, the elapsed times of SPCG (pipe) and SPCG (p-scheme) are about 23 msec and about 13 msec, respectively. ISPCG is somewhat worse than NoZSolve, but both require the almost same time to reduce the residual to $1.0 \times 10^{-15}$.

![Fig. 3 Total performance (size = 96³).](image-url)
An Incomplete Splitting-up Conjugate Gradient Method for Parallel Computing

Fig. 4  Total performance (size = $192^3$).

Fig. 4 shows the total performance when the data size is $192^3$. In this case, NoZSolve exhibits the best performance, too. When the residual is reduced to $1.0 \times 10^7$, NoZSolve achieves the speedup of about 18 compared with SPCG (1 proc). Other results are almost same as the previous case, except for ISPCG. Namely, SPCG (p-scheme) exhibits the second best performance. The reason why the total performance of ISPCG is not good is that the speed of convergence is bad until $1.0 \times 10^8$.

According to the results of the experiments, SPCG (p-scheme) does not overcome NoZSolve, but the convergence performance of incomplete SPCG methods may be degraded when other parameters are taken. When the bandwidth of network of multicomputers is fast, the elapsed time of SPCG (p-scheme) can be improved, and thus the total performance of SPCG (p-scheme) may be comparable or superior to that of NoZSolve.

5. Conclusion

This paper describes several variants of the SPCG method suitable for parallel computing and evaluates the performance and the speed of convergence on a distributed-memory multicomputer.

Among the variants, one of incomplete SPCG method, which does not carry out one of three tridiagonal matrix solvers, achieves the best performance, and this method is about 20 times faster than one-process version of the SPCG method on 32 CPU cores of SX-ACE supercomputer.

In the future, we will compare the performance of the SPCG method with that of a parallelized ICCG method and we will also evaluate our methods on other multicomputers in which the speed of network is faster than the SX-ACE.

Acknowledgements

The author is grateful to Professor Tatsuo Nogi of Kyoto University for helpful discussions, also has received generous support from Professor Shinji Odanaka of Osaka University for using SX-ACE supercomputers. The author would like to express his gratitude to both professors.
This work was supported by JSPS KAKENHI Grant Number 15K00501. This research was also supported in part by MEXT, Japan.

References


